CM50109

FORMAL METHODS AND PROGRAMMING

COURSEWORK 2

EngD GROUP 5

DOCUMENT 1

Matthew Thompson (Group Leader)

Milto Miltiadou

Hashim Yaqub

1) Project Overview

Introduction
The object of this assignment was to create a digital implementation of the well-known board game 'Monopoly'. The following documents give a detailed description of the software production process, and an overview of how to use the game.

Documentation Structure

1) Document 1
    1. Requirements Specification and Analysis
    2. Problem Analysis and Design
    3. Development Methodology
    4. Brief User Guide

2) Document 2
    1. Testing
    2. Limitations and Further Implementation
    3. Maintenance Guide
    4. Implementation Code

3) Project Diaries

4) Meeting Minutes

Group Member Task Allocations

1) Matthew Thompson (Group Leader)
    1. In charge of Testing and Debugging (documentation)
    2. Worked on some design documentation
    3. Wrote Agile Programming Log
    4. Dealt on Player, Game and Dice classes in C++

2) Milto Miltiadou
    1. In charge of programming.
    2. Majority of implementation, and supervision of other's code
    3. Limitations and Further Work Section
    4. Doxygen Documentation#

3) Hashim Yaqub
    1. In charge of documentation
    2. Any classes dealing with Cards
    3. Worked with Matthew on some testing
    4. Majority of documentation

2) Software Requirements Specification and Analysis

Introduction

This is digital recreation of the board game Monopoly. It is to be implemented in C++ using a using a console interface for game-play . The object of the game is to by property and strategically manage finances. The object of the game is to bankrupt the other players.

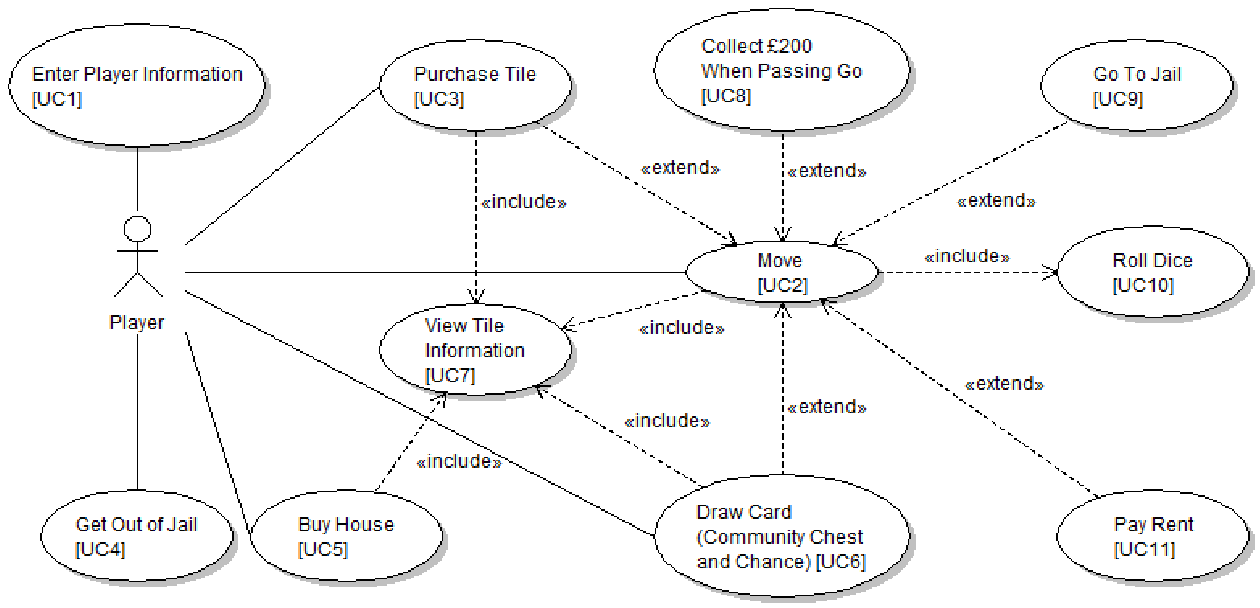
Rules of the game

1) At the beginning of the game, each player rollers the dice. The player with the highest number goes first.
2) When a player moves to a property tile, player has the option to buy the property.
3) Once player owns a property tile, any other player who lands pays that player rent according to stated value.
4) Once a player owns a group of properties (indicated by colour), they can not only charge higher rent, but also are allowed to build houses and hotels on a property.
5) Each house increases rent.
6) A hotel can only be built ones 5 houses are built on ONE property.

Other Game Components include:-

1) The ‘Go’ tile is where each player starts from. Every time this tile is passed, each player receives £200.
2) ‘Community Chest’ and ‘Chance’ tiles are special cases. When a player lands on one of these, they are randomly assigned a task, which includes receiving or giving money, going to jail and being free from jail
3) The ‘Go To Jail’ tile penalises a player by putting them out of the game for up to 3 turns, unless the player pays a £50 fee.
4) ‘Water Works’ and ‘Electric Company’ are special tiles where the owner can charge rent to the target player according to the dice roll.
5) There are 4 Railway cards, which increase with rent depending on how many railways the player owns.
6) ‘Income Tax’ is a tile where the player is forced to pay £100 to the bank.

Use Case Diagram.



Use Case UML diagram showing the different actions a player takes throughout the process of the game. It is then followed by the flow descriptions detailing the different interactions.

UC1:- Enter Player Information – Flow Events
Preconditions: None
Main Flows: At the start of the game, console prompts for users to enter how many players there are. User then enters a value between 2 or 6. User then enters name for each of the players. Once this is done, each player roles the dice. The player with the highest number goes first.
Subflows: None
Alternative Flows:

- If the player does not enter an integer, or a number between and including 2 and 6, then the user is prompted to enter value again.
- If the player does not enter a string for the player name, they are prompted to enter again.

UC2:- Move – Flow Events
Preconditions: All player information has been accepted, and player order has been determined.
Main Flows: Each player is moved according to the value the dice returns after they roll it. Each player takes turns to roll the dice. Each player can also be moved depending on certain circumstances, such as landing on the 'Go To Jail' tile, or drawing a card from 'Community Chest' which commands the player to move to a specified tile, e.g. 'Advance To Old Kent Road'.
Once the player has been moved to a tile, the Tile's information is then displayed.
Subflows: Once player has been moved to a cell, an action or set of actions are taken depending on the type of cell:-

- Purchase Tile [UC3]
- Collect £200 when Passing Go [UC8]
- Draw Card(Community Chest and Chance) [UC6]
- Pay Rent [UC11]

Alternative Flows: None
UC3:- Purchase Tile– Flow Events
Preconditions: The player has rolled the dice and landed on a tile which is available for purchase
Main Flows: The information of the cell is displayed. The player is then prompted with a 'yes or no' question about whether they would like to purchase the property. If yes is selected, the player

pays the amount of money indicated by the tile information to the bank, and the property belongs to that player. If they select no, the property remains available and the player's turn is ended.
Sub-flows: None
Alternative Flows: If the player does not have enough money, a message stating so is presented on the console, and the turn is ended.

UC4:- Get Out Of Jail – Flow Events
Preconditions: The player must already be in jail before this event can be executed. A 'Get Out Of Jail Free' card can be acquired from one of the 'Draw Card' tiles.
Main Flows: The player can get out of jail either by using the 'Get Out of Jail Free' card, or they can pay £50 to get out.
Sub-flows: None
Alternative Flows: The player cannot afford the £50 and does not have a card to get out. Therefore they are out of the game.

UC5:- Buy House– Flow Events
Preconditions: The player must do this BEFORE they roll the dice. They can only purchase houses (and hotels) for properties for which they have all of the matching colour groups.
Main Flows: At the start of the players turn, if they have a group of properties, they are prompted as to whether they would like to build any houses. The player is able to build one house at a time on each of the properties. Each property has 5-house limit, where 5 houses becomes a hotel.
Sub-flows: None
Alternative Flows: If player does not have enough money, they are told so in a message, and nothing happens. The same is done for when the player already has 5 houses on a property.

UC6:- Draw Card(Community Chest and Chance)– Flow Events
Preconditions: The player has rolled the dice and has landed on either the 'Community Chest' or 'Chance' Tile.
Main Flows: When the player lands on one of the tiles, and 'task card' is drawn at random. This is presented in the form of instruction (e.g. You have been caught drink-driving, pay £150 fee) or a statement (You have prevented a Warp Core breech, have £10 Mr LaForge).
Sub-flows: None
Alternative Flows: None

UC7:- View Tile Information– Flow Events
Preconditions: Tile information can be viewed when a player lands on it, or when a player would like to build property on it.
Main Flows: Tile information is presented, displaying attributes such as name, rent price, who owns it (or if it is available) etc.
Sub-flows: The property information is updated and re-presented when a player buys the property, puts a house on it, or when property is disowned by player due to bankruptcy.
Alternative Flows: None

UC8:- Collect £200 When Passing Go– Flow Events
Preconditions: The player passes the Go tile after dice roll.
Main Flows: Regardless of where the player lands afters passing the 'Go' tile, they receive £200 before any further transactions are made e.g. because they have landed on a card for which they must rent for.
Sub-flows: None

Alternative Flows: *I*n some circumstances the player does not receive £200. These are:

- When player is sent to jail (by tile or by drawing card)
- When the player is asked to move back a certain amount of spaces

UC9:- Go To Jail– Flow Events
Preconditions: The player lands on the 'Go To Jail' tile, or the player draws a 'Go To Jail' card
Main Flows: Player is moved to the 'Jail' tile where they remain until the player pays £50, or they use a 'Get Out Of Jail Free' card.
Sub-flows: None
Alternative Flows: None

UC10:-Roll Dice– Flow Events
Preconditions: It's the player's turn.
Main Flows: The console prompts player to 'Press Enter to Roll'. Random number is generated by the Dice. This number can be any integer between 2 and 12 (2 dice).
Sub-flows: None
Alternative Flows: None

UC11:-Pay Rent– Flow Events
Preconditions: Player rolls and lands on tile which is owned by another player
Main Flows: The player has to pay a certain amount of rent to the player who owns the property
Sub-flows: The amount of rent paid is dependent on the following factors:-

- The individual rent value of the property
- The effected rent value of a property when the owner has the whole set of properties
- The effected rent value of a property when houses (and hotel) are built on it
- If it is a utility ('Water Works' and/or 'Electric company') then the amount of rent to pay is dependent on the value of the dice roll.

Alternative Flows: If the player does not have enough money, they must declare bankruptcy.

Non-Functional Requirements

1) **Performance**
    a. **Environment:-** The Monopoly game is designed to be executed as a text-based user-driven console application. Response time is expected to be minimal with a very short delay between user input and console output. Each 'move' is represented by text messages indicating each key even in the game, e.g. when a player moves to a new square, and has to pay rent to another player.
    b. **Game Updates:-** Due to the completely event-driven nature of the game, each update is expected to be almost immediate for each immediate.
2) **Usability**
    a. Each player is presented with the options as and when they are needed. Each option is decided with a keyboard input specified by the current instruction.
    b. It is all Keyboard interaction, where the amount of keys is kept to a minimum as much as possible. The most typing any player has to do is when entering their name.
    c. Other than that, it is mostly 1 or 2 character inputs

3) **Portability**
    a. **Implementation:-** The game is implemented in C++ on Unix based systems. It only utilises built-in libraries which would allow for greater cross compatibility. No graphics libraries are used as it is purely text-based.
    b. **Compilation:-** It is compiled using the GNU C Compiler

4) **Operating Constraints**
    a. Needs between 2 to 6 players to properly play it.
    b. Requires only once machine or terminal to play on. Cannot be played across multiple terminals (e.g. LAN).
    c. Need to have C++ (build essentials) installed on machine in order to play

5) **Modifiability**
    a. The **board** itself can be modified by editing the 'Board' File. All but the 'Go', 'Go To Jail', 'Jail', 'Community Chest' and 'Chance' tiles can be modified. Within the 'Board' file, each line is written with a particular format depending on the type of tile it is. As long as the user confirms to the customisation guidelines (see user manual), then they can make their own properties with different rent values and names etc. Generally, the format is POSITION NUMBER, TYPE FLAG, PROPERTY VALUE, RENT VALUES, GROUP, NAME.
    b. **Community Chest** and **Chance** cards can also be customised in the same way as the board. The lines in both the 'Community Chest' and 'Chance' file follow a simpler structure. The format goes FLAG, 0-to-2 PARAMETERS, INSTRUCTION. See user manual for proper formatting.
    c. **Modifying Card Manager:-** The Card Manager can be modified to accept different/new types of cards. For example, if someone wanted to a make a card which causes the player to 'Go To Jail' and lose £300 at the same time, they would have to create their own flag. In order for card manager to recognise this, they would have to include it in the 'if' statement which checks for each flag.
    The user might want this flag to be 'jl'. They would have to also create a class to deal with this operation.
    d. Number of players can also be modified from 6 to e.g. 8 players, as long as there are at least 2 players.

6) **Security Considerations**
    a. When it is a player’s turn, that player cannot in anyway directly access the statistics or attributes of any other player.
    b. An Object Oriented approach allows for abstract separation and integrations of key functionality without comprising any security. It is ensured that any methods or functions exclusive to an object or class remain private.

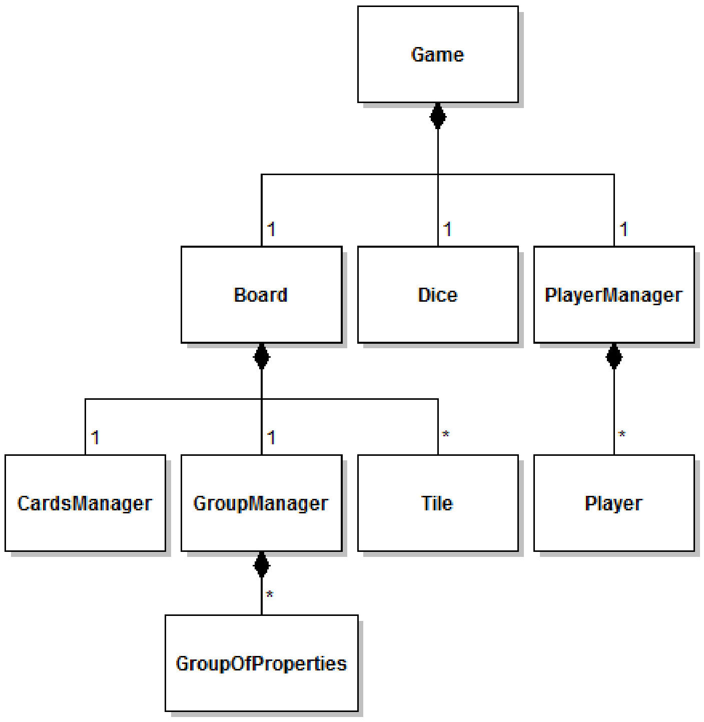
7) **Error Handling**
    a. If the player enters an invalid value, the system waits until a valid input has been entered.
    b. Invalid entries in the Board, Community Chest or Chance file, are flagged up as the game is loading, at which point the game will exit. So if someone for example uses the ‘g’ flag (for when a player receives money) in the Chance file, but they don’t have an integer parameter afterwards, this will be flagged up
    c. If a player enters an empty string when entering player name, they will be prompted to enter a name.

Problem Analysis and Design

General Overview

Introduction:-

The problem was abstracted away into six main categories of class: Game, Board, Tile, Dice, Card and Player. There are also 'manager' classes to take care of the objects for each class.



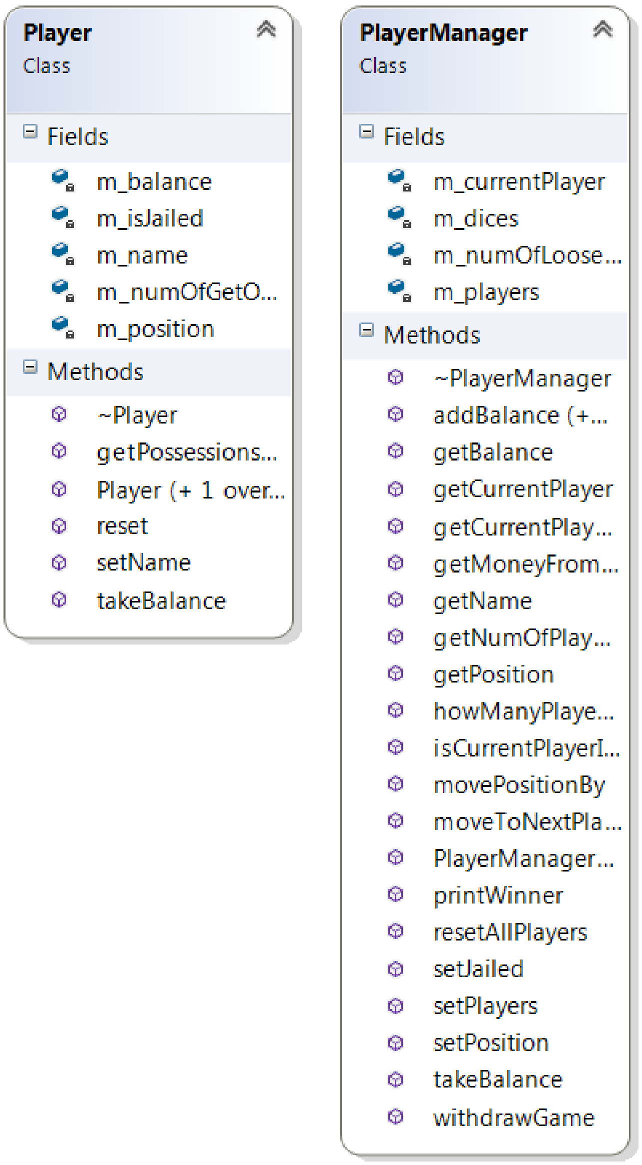
Class Breakdown:-

When the game is first started, an instance of the Game class is created. This object then creates instances of Board, Player and Card to 'populate' the game. Each Tile, Player and Card object describes one of many items, so vectors (arrays) containing collections of objects are kept in the Manager classes. So, for example, a vector (array) of Players is kept and managed in the PlayerManager class and a vector of Cards is stored and managed from the CardsManager class.

Player Manager

Introduction:-

At the start of the game, the system asks the user how many people are playing, and for their names. These actions are performed by the Game object and are then used by the Player Manager class to create Player instances.



Class Breakdown:-

1) The Game class is not really a proper object as such, more just a wrapper for the main game loop. When the program is first run, it simply sets up the game by creating PlayerManager and Board objects. A Dice object is also created.
2) The PlayerManager object asks the user for the number or players and their names. This information is used to populate an array of Player objects.

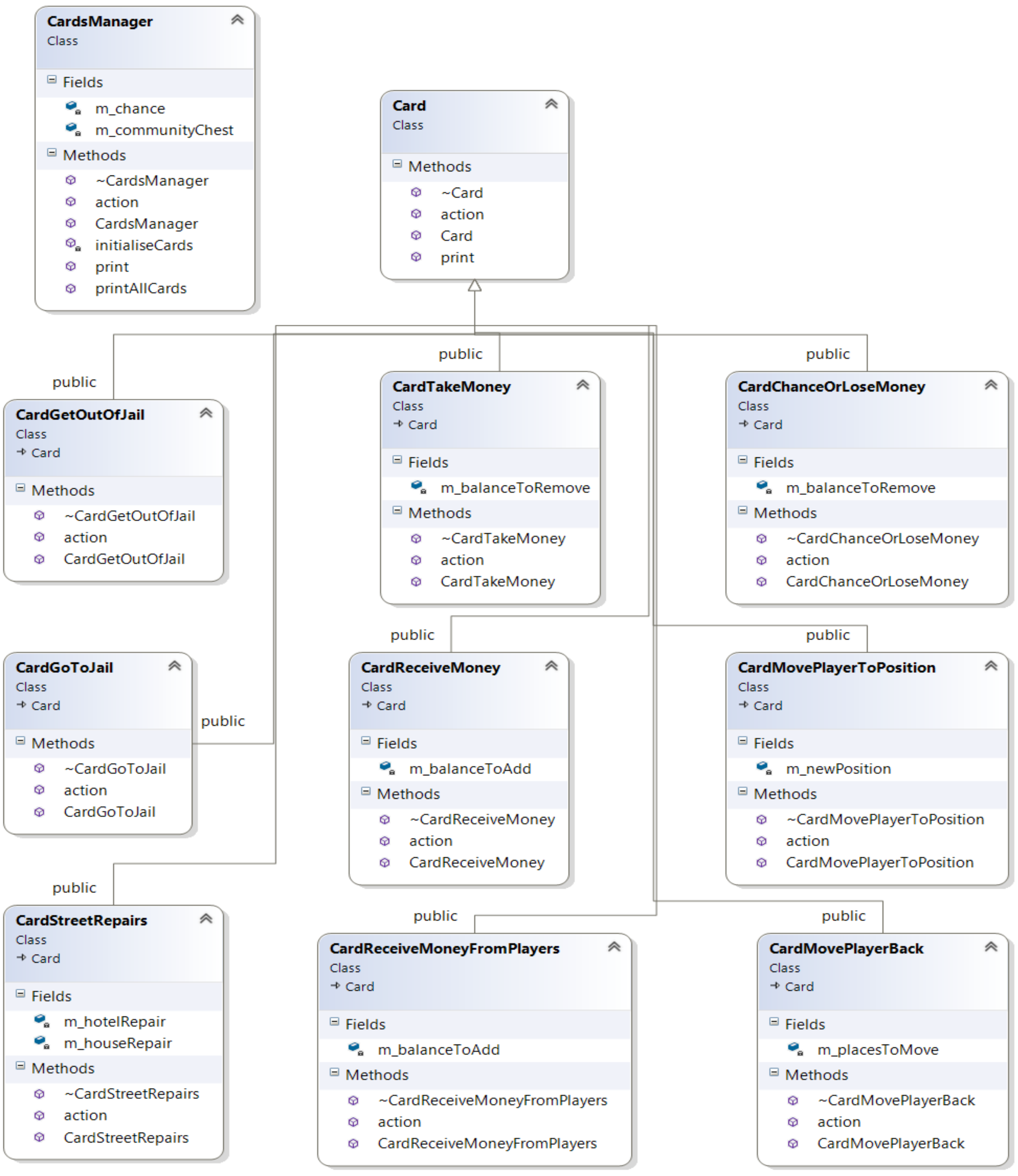
3) The Board object reads tile details in from a text file and populates an array of Tile objects. Each tile object contains information such as the group it belongs to (as handled by the GroupManager class) and the action performed when the player lands on it.
4) Once the Player and Board objects have been initialised and put into arrays, the Game class calls a takeTurn method. This method is called each time a player takes a turn. It rolls the dice, moves the player and takes actions specified by the tile a player lands on.
5) Every time a player takes a turn, the Player Manager class manipulates their player object, changing its properties and calling methods.

Cards Manager

Introduction:-

One of the key components of the Monopoly game is ‘Drawn Card’ function (see Use Case 6). When a player lands on the ‘Community Chest’ or ‘Chance’ tile, then the Draw Card action is called into play. The CardsManager class deals with the actions involved in this process.

Class Breakdown:-



1) Within the constructor of '**Cards Manager'**, during initialisation, two files are read into 2 string vectors; the Chance and Community Chest text files.
2) At the next stage, each string entry in the community chest and chance vectors are further parsed and segmented up so as to be made into a **Card** object
3) There are 9 types of card altogether. In the text files, these types are denoted by flags, which are the characters at the beginning of the line. Each type of card takes 0 to 2 parameters. These parameters are denoted straight after the flag. Finally each Card has an instruction. Below is an example of

```
h 40 115 You Are Assessed for Street Repairs. £40 per House, £115 per Hotel
```

4) When the line from the vector is passed, the first 'word' is parsed into a temporary string, which is then matched with a set of comparison conditions i.e. the flags.
5) As shown in the class diagram above, there are 9 classes which inherit from **Card.** These represent the different types of card and influence and manipulate the players accordingly. Each of these classes contains an 'which take 0 to 2 parameters. They also contain an action with takes a reference to the players as a parameter.
6) The Player's statistics are manipulated via the **PlayerManager**, through the **action**.

FLAGS	**Description**	**Class**
b	Lose Money or Pay Fee	`new CardChanceOrLoseMoney(moneyToRemove)`
f	Get Out of Jail Free	`new CardGetOutOfJail()`
g	Player Receives Money	`new CardReceiveMoney(moneyToAdd)`
gp	Player Receives Money from Other Players	`new CardReceiveMoneyFromPlayers(moneyToAdd)`
h	Pay for Houses and Hotels	`new CardStreetRepairs(house, hotel)`
j	Go To Jail	`new CardGoToJail()`
l	Player Loses Money	`new CardTakeMoney(moneyToRemove)`
m	Move Player to Position	`new CardMovePlayerToPosition(boardPosition)`
mb	Move Player back an amount of Spaces	`new CardMovePlayerBack(spacesToMove)`

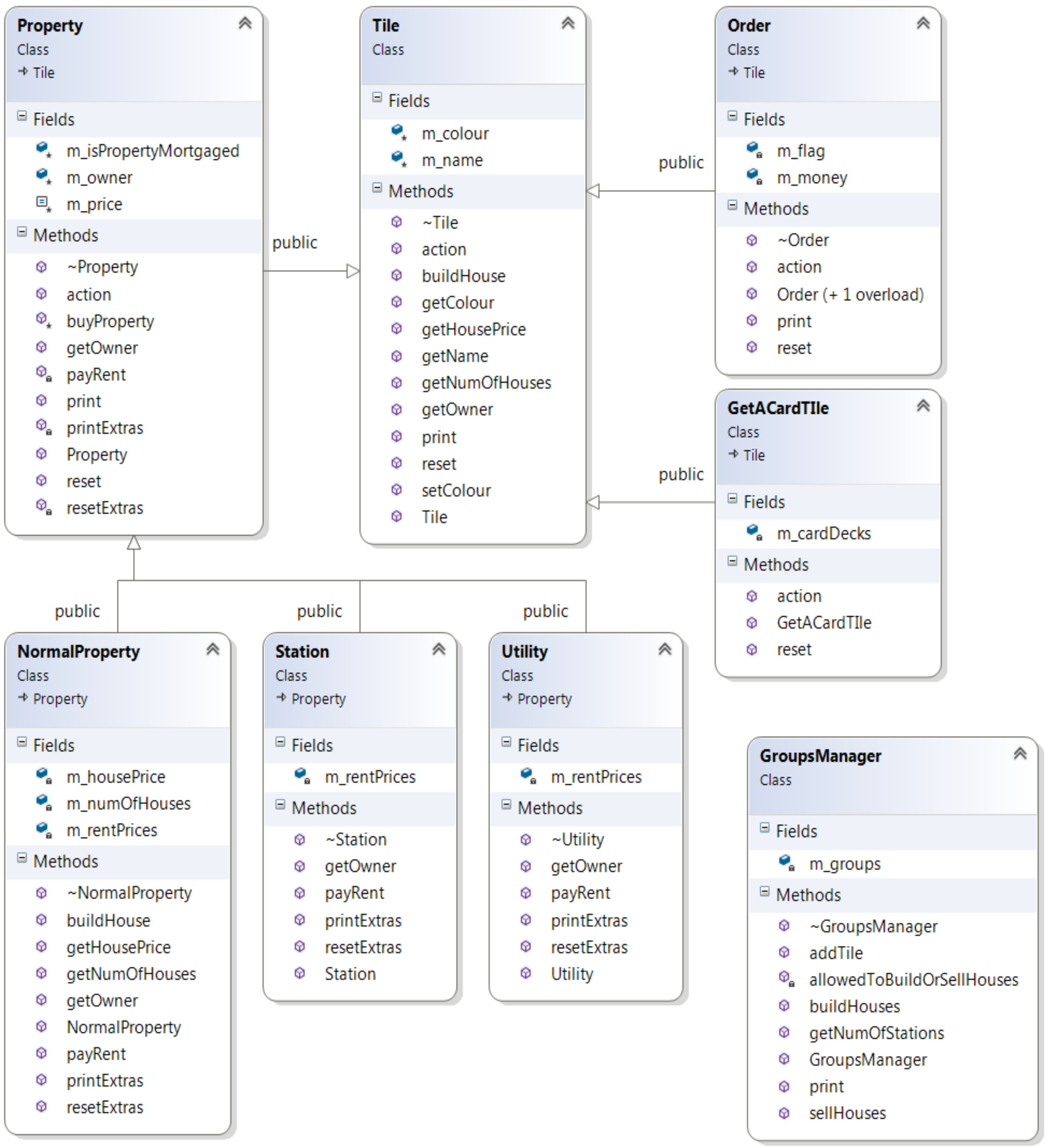
7) Once all of the cards have been created, they are stored into a **vector of cards** for 'Community Chest' and 'Chance'.
8) **CardsManager** itself has an **action**. This is called whenever a player lands on the Community Chest or Chance tile. The name of the tile is passed to this action, as well as the reference to the player who landed on it.
9) A random number is then generated and passed into the appropriate **card vector,** and the chosen card's action is performed.

Tile

Introduction:-

Each item on the Monopoly board is represented by a **Tile.** Each is separated into types which are defined in the Board text file. The most prominent type of tile are the **Normal Property** tiles. Each of these belong to a group, denoted by colour. Each group consists of 2 or 3 properties. **Group Manager** is used to manage the relations between these tiles, as some operations e.g. building houses cannot be performed by players until the player owns all of the properties in a group.

Class Breakdown:-



1) When the game is loaded, the first component to be loaded is the **board**. The board items (tiles) are defined in the **Board text file**.
2) The structure of each line is such that they are assigned CELL NUMBER, FLAGS, PARAMETERS and NAME.
3) There are **FOUR** main types of tiles
   a. Order:- These are the tiles which are integral to the game (and should not be changed). These include the GO, JAIL, FREE PARKING, GO TO JAIL, SUPER TAX, COMMUNITY CHEST and CHANCE tiles.
   b. Normal Property:- These are the majority tiles. Each of these tiles are part of a group of properties. It's unique feature is that once the player owns all of the properties in a group, they are able to build houses and hotels on them to increase the amount of rent they receive
   c. Station:- Railway Stations are all the same price. However the rent chargeable by the owner is dependent on how many other stations they own.
   d. Utility:- Utilities (WATER WORKS and ELECTRIC COMPANY) are the same price. However the rent chargeable by the owner is dependent on how many other stations they own. It is also dependent on the value of the second dice roll.
4) Normal Property, Station and Utility all inherit from the **Property** class. **Property** keeps track of the player who owns the property, and deals with calculating the amount of rent to charge any opposing player who lands on the tile.

FLAGS	**Description**	**Class**
p n	Normal Property	`new NormalProperty(name,price,housePrice,rentPrices)`
c	Cards	`new GetACardTIle(name,m_cards)`
o p	Tax Tiles	`new Order(name,"p",money)`
o j	Go To Jail	`new Order(name,flag)`
o f	Free Parking	`new Order(name,flag)`
p s	Railway Station	`new Station(name,price,rentPrices)`
p u	Utility	`new Utility(name,price,rentPrices)`

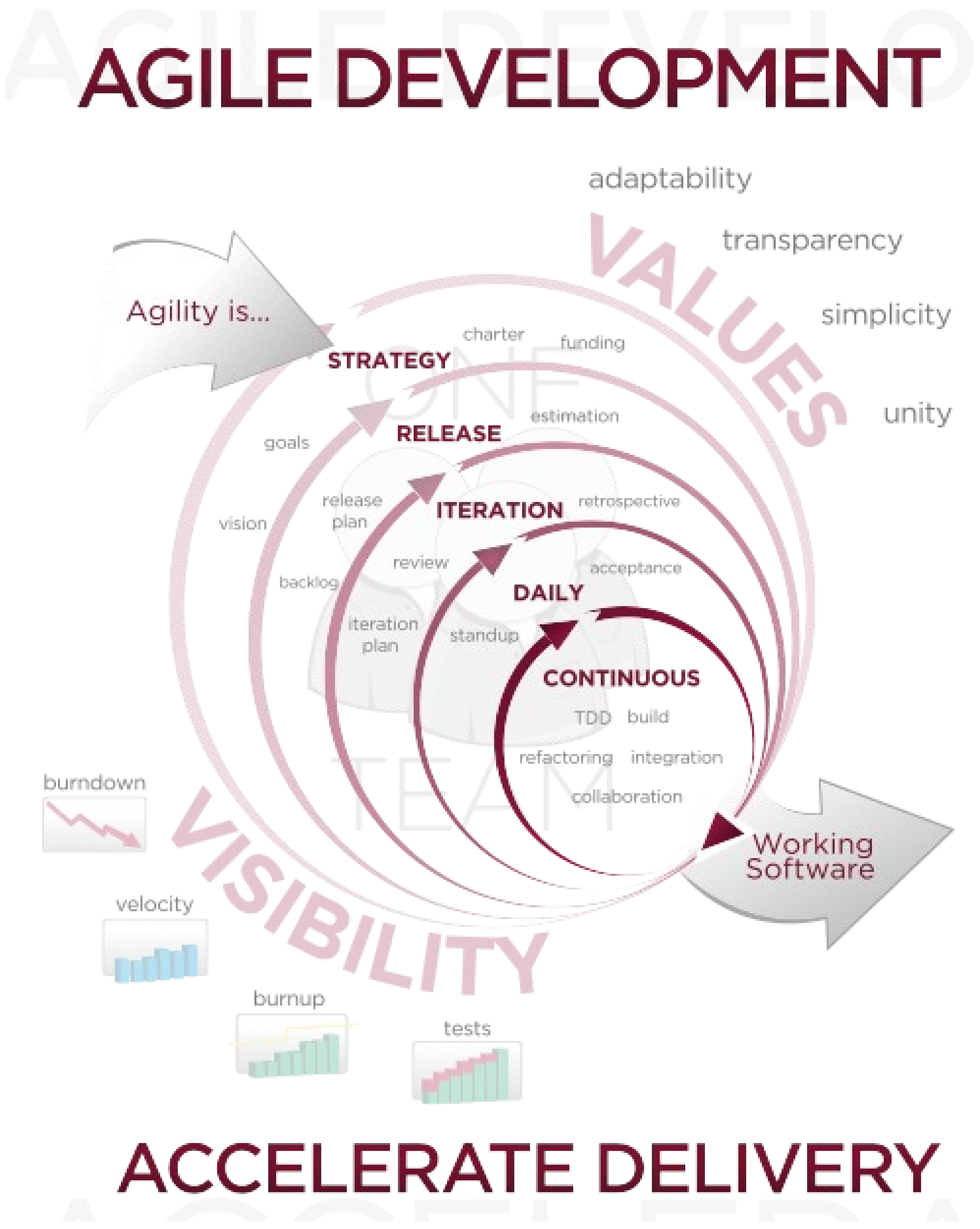
5) All of the 'Order' tiles are then stored into a vector of tiles

The remaining tiles which inherit from **Property** (Normal Property, Station and Utility) are also stored within the same vector of tiles. However, they are also stored into their appropriate groups through the **GroupsManager** class. The **colour** (group) attribute is also set e.g.

```
m_groups.addTile(colour,m_tiles[counter]);
m_tiles[counter]->setColour(colour);
```

3) Development Methodology

Approach taken and rationale



(image from Wikimedia commons, used under a CC-BY-SA 3.0 license)

An Agile approach to software development was taken towards developing this system. This means that we took a focus on getting a basic working version of the program together as soon as possible, then iteratively adding new features to it until the design specification was met.

The Agile Manifesto

Agile puts a focus on:

1) Individuals and interactions
2) Working software
3) Customer collaboration
4) Responding to change

Though we did not have any customers to collaborate with or change to respond to, we thought that it would be a good idea to adopt an Agile approach for its other main benefits: the focus on people and working software. Having a working system early on in the development lifecycle allowed us to reason better about how to build the system from a simple one to a complex one as features were added.

We decided to take an Agile approach to development because of the following benefits:

10) Pair programming suited our team well. We had a variety of skillsets in our team, with some members having a string C++ background, and others having a better grasp of Object Oriented programming through previous exposure to Java. Pair programming allowed team members to combine their skills.
11) Use of version control allowed each team member to add changes quickly so that the other team members could examine the new program and advise on it. We used the Git version control system on this project.
12) Having a working version of the program at all times, no matter how simple, gave us confidence that we would always have something to hand in, no matter how much progress was made towards fulfilling all requirements.

Description of program versions

Version 0.1
We have three main classes: Board, Tile and Player. Tile inherits from board. When the program is run, the Board class populates arrays with empty Tile and Player objects. At the moment, all class methods and constructors are empty placeholders.

Version 0.2
The game loop was implemented in this version. This simply starts the game, asks how many people are playing, then asks for their names. Player objects are then created based on this information.

Version 0.3
Added different kinds of tiles and properties. NormalProperty, Station and Utility all inherit from tile, but have different properties and methods. Dice class was added.

Version 0.6
We implemented buying and selling of houses on properties in this version. GroupManager class was added to facilitate this.

Version 0.8
Major changes made to the way the system prompts for user input via the command line. User can

now buy and sell houses.

Version 1.0
Minor bugfixes based on testing were implemented. Extra commenting added.

4) Brief User Manual

Running the Game

a. This version of Monopoly is a text-based C++ implementation of the game which is run entirely on the console.
b. To run the game, in the terminal, navigate to the ‘Monopoly’ folder and run the command **./Monopoly**
c. If you would like to compile/recompile it first before playing (in case of modifications), then use the command **make** (for Clang compiler) or **make gcc** (for GNU C Compiler).

Setting Up

a. When the game runs, you will be prompted to enter the number of players for game (between 2 or 6)
b. You will then be asked to enter all of the player names
c. You are now ready to play.